IEEE P802.11
Wireless LANs

|  |
| --- |
| Resolution of Security Comments from 1st Sponsor Ballot Recirculation |
| Date: 2011-01-14 |
| Author(s): |
| Name | Affiliation | Address | Phone | email |
| Dan Harkins | Aruba Networks | 1322 Crossman ave, Sunnyvale, CA., USA | +1 408 227 4800 | dharkins at arubanetworks dot com |
|  |  |  |  |  |

Abstract

This document proposes resolution to CIDs 1002, 1005, 1006, 1008, 1011, 1015, 1016, 1018, 1019, 1020, 1022, 1023, 1033, 1043, 1052, and 1326 from the First Recirculation of the Sponsor Ballot.

***Add the word “Yes” into the Extensible column of table 7-26 in section 7.3.2.0a for the row whose element is MIC.***

***Remove the word “Yes” from the Extensible column of table 7-26 in section 7.3.2.0a for the row whose element is Mesh Peering Management.***

***Modify section 8.1.3 as indicated***

* RSNA Establishment

Change the contents of item b) and c) in 8.1.3 as follows:

* If an RSNA is based on a PSK or password in an ESS, the SME establishes an RSNA as follows:
* It identifies the AP as RSNA-capable from the AP’s Beacon or Probe Response frames.
* If the RSNA-capable AP advertises support for SAE authentication in its Beacon or Probe Response frames, and the STA has a group defined in the dot11RSNAConfigDLCGroupTable and a password for the AP in the dot11RSNAConfigPasswordValueTable, the STA shall invoke SAE authentication and establish a PMK. If the RSNA-capable AP does not advertise support for SAE authentication in its Beacon and Probe Response frames but advertises support for the alternate form of PSK authentication (see 5.8.2.2 (Alternate Ooperations with PSK)), and the STA also supports the alternate form of PSK authentication, the STA may ~~It shall~~ invoke Open System authentication and use the PSK as the PMK with the key management algorithm in step 4) below.

***Modify section 8.2a.4 as indicated***

* Finite cyclic groups

SAE uses discrete logarithm cryptography to achieve authentication and key agreement. Each party to the exchange derives ephemeral public and private keys with respect to a particular set of domain parameters that define a finite cyclic group. Groups can be based on either Finite Field Cryptography (FFC) or on Elliptic Curve Cryptography (ECC). Each component of a group is referred to as an “element.” Groups are negotiated using an identifying number from a repository maintained by IANA as “Group Description” attributes for IETF RFC 2409 (IKE). The repository maps an identifying number to a complete set of domain parameters for the particular group. For the purpose of interoperability, conformant STAs shall support group nineteen (19), an ECC group defined over a 256-bit prime order field.

More than one group can be configured on a STA for use with SAE by using the dot11RSNAConfigDLCGroup table. Configured groups are prioritized in ascending order of preference. If only one group is configured it is, by definition, the most preferred group.

NOTE—The preference of one group over another is a local policy issue.

SAE uses two arithmetic operators defined for both FFC and ECC groups, an operation that takes two elements to produce a third (called the “element operation”), and an operation that takes one element and one scalar value to produce another element (called the “scalar operation”). The convention used here is to represent group elements in uppercase bold italic(Ed) and scalar values in lowercase italic(Ed). The element operation takes two elements, *X* and *Y*, to produce a third element, *Z*, and is denoted *Z* = elem-op(***X***,***Y***) while the scalar operation takes a scalar, *x*, and an element, *Y*, to produce another element *Z* and is denoted *Z* = scalar-op(*x*,***Y***).

scalar-op(*x*,***Y***) can be defined as successive iterations of elem-op(***Y***,***Y***). That is, it is possible to define scalar-op*(*1,***Y***) = ***Y*** and for *x* > 1, scalar-op(*x,* ***Y***) = elem-op(scalar-op(*x-*1,***Y***),***Y****)*. The specific definition of elem-op(***X****,****Y***) depends on the type of group, either ECC or FFC.

*Modify section 8.2a.4.1.1 as indicated*

* ECC group definition

ECC groups used by SAE are defined by the sextuple (p, a, b, G, r, h) where *p* is a prime number, *a* and *b* specify the elliptic curve defined by the equation, *y*2 = *x*3 + *ax* + *b* modulo *p*, ***G*** is a generator (a base point on the elliptic curve), *r* is the prime order of ***G***, and a co-factor *h*. Elements in ECC groups are the points on the elliptic curve defined by their coordinates—(*x*, *y*)—that satisfy the equation for the curve and the identity element, known as the “point at infinity.”

The IANA registry used to map negotiated numbers to group domain parameters includes definitions of some ECC groups defined over a characteristic 2 finite field. These groups shall not(CID194) be used with SAE. In addition, some elliptic curves have a co-factor greater than one (1). These groups also shall not(CID194) be used with SAE. Only ECC groups defined over an odd prime finite field with a co-factor equal to one (1) shall be(CID194) used with SAE.

The element operation in an ECC group is addition of two points on the curve resulting in another point on the curve. For example, point ***X*** is added to point ***Y*** to produce point ***Z***:

 ***Z*** = ***X*** + ***Y*** = elem-op(***X***,***Y***)

The scalar operation in an ECC group is multiplication of a scalar value by a point on the curve, or the repetitive addition of a point on the curve with itself a certain number of times, resulting in another point on the curve. For example, the point ***G*** is multiplied by the scalar *q* to derive the point ***Q***:

 ***Q*** = *q****G***(Ed) = scalar-op(*q*,***G***)

SAE requires an additional operation, inverse(), to produce the inverse of a point on an elliptic curve. A point on an elliptic curve is the inverse of a different point if their sum is the “point at infinity.” In other words: ***Q*** + inverse(***Q***) = point at infinity

ECC groups make use of a mapping function, F, that maps an element from the group to a scalar value. For ECC groups, function F shall be instantiated by returning the x-coordinate of a point — i.e., if ***P*** = (*x*, *y*) then F(***P***) = *x*.

NOTE—SAE protocol operations preclude function F from ever being called with the identity element, e.g. the “point at infinity”.

***Modify 8.2a.4.1.2 as indicated***

* Generation of the Password Element with ECC groups

The Password Element of an ECC group (***PWE***) shall be generated in a random hunt-and-peck fashion. A counter, represented as a single octet and initially set to one (1), is used with the peer identities and the password to generate a password seed. The password seed shall then be stretched using the key derivation function (KDF)(Ed) from 8.5.1.5.2 to the bit length of the prime number from the group definition with the Label of “SAE Hunting and Pecking” and the Context being the prime. If the resulting password value is greater than or equal to the prime, the counter shall be incremented, a new password seed is derived and the hunting-and-pecking shall continue. If the password value is less than the prime it shall then be used as the x-coordinate of a curve and the equation for the curve shall be checked to see if a solution for *y* exists. If no solution exists, the counter shall be incremented, a new password-seed shall be derived and the hunting-and-pecking shall continue. If a solution exists, there will be two possible values for *y*. The password seed is used to determine which one to use. If the LSB of the password seed is equal to the LSB of y returned as the solution to the quadratic equation then the ***PWE*** shall be (*x*, *y*) otherwise the ***PWE*** shall be (*x*, *p – y*).

NOTE—the probability of finding ***PWE*** is (*p*/*r*)n after *n* iterations of the “hunting and pecking” loop which rapily approaches zero (0) as *n* increases.

Algorithmically this process can be described as follows:

 *found* = 0;

 *counter* = 1

 z = len(*prime*)

 do {

 *pwd*-*seed* = H(MAX(STA-A-MAC, STA-B-MAC) || MIN(STA-A-MAC, STA-B-MAC),

 *password* || *counter*)

 *pwd*-*value* = KDF-z(*pwd*-*seed*, “SAE Hunting and Pecking”, *prime*)

 if (pwd-value < *prime*)

 then

 *x* = *pwd*-*value*

 if the equation *y*2 = *x*3 + *ax* + *b* has a solution *y*

 then

 determine the solution, *y*, to the equation *y*2 = *x*3 + *ax* + *b*

 if LSB(*pwd-seed*) = LSB(*y*)

 then

 ***PWE*** = (*x*, *y*)

 else

 ***PWE*** = (*x*, *p – y*)

 fi

 *found* = 1

 fi

 fi

 *counter* = *counter* + 1

 } while (*found*=0)

***Modify 8.2a.5.2 as indicated***

* PWE and secret generation

Prior to beginning the protocol message exchange, the secret element ***PWE*** and two secret values are generated. First, a group is selected, either the most preferred group if the STA is initiating SAE to a peer, or the group from a received Commit Message if the STA is responding to a peer. The ***PWE*** shall be generated for that group (according to 8.2a.4.1.2 (Generation of the Password Element with ECC groups) or 8.2a.4.2.2 (Generation of the Password Element with FFC groups), depending on whether the group is ECC or FFC, respectively) using the identities of the two STAs and the configured password.

After generation of the ***PWE***, each STA shall generate a secret value, *rand*, and a temporary secret value, *mask*, each of which shall be chosen randomly such that 1 < *rand*, *mask* < *r*, the order of the group. The values *rand* and *mask* shall be random numbers produced from a quality random number generator. They shall never be reused on distinct protocol runs.

***Modify 8.2a.5.4 as indicated***

* Processing of a peer’s Commit Message

Upon receipt of a peer’s Commit Message both the scalar and element shall be verified.

If the scalar value is greater than zero (0) and less than the order, *r*, of the negotiated group, scalar validation succeeds, otherwise it fails. Element validation depends on the type of group. For FFC groups, the element shall be an integer greater than zero (0) and less than the prime, and the scalar operation of the element and the order of the group, *r*, shall equal one (1) modulo the prime *p*. If either of those conditions does not hold element validation fails; otherwise, it succeeds. For ECC groups, both the x- and y-coordinates of the element shall be integers less than the prime, *p*, and the two coordinates shall produce a valid point on the curve using the group’s curve definition. If either of those conditions does not hold, element validation fails; otherwise, element validation succeeds.

***Modify section 8.2a.6 as indicated***

8.2a.6 Anti-Clogging Tokens

A STA is required to do a considerable amount of work upon receipt of a Commit Message. This opens up the possibility of a distributed denial-of-service attack by flooding a STA with bogus Commit Messages from forged MAC addresses. To prevent this from happening, a STA shall maintain an *Open* counter in its SAE state machine indicating the number of open and unfinished protocol instances (see 8.2a.5.1). When that counter hits or exceeds dot11RSNASAEAntiCloggingThreshold, the STA shall respond to each Commit Message with a rejection that includes an Anti-Clogging Token statelessly bound to the sender of the Commit Message. The sender of the Commit Message must then include this Anti-Clogging Token in a subsequent Commit      Message.

The Anti-Clogging Token is a variable length value that statelessly binds the MAC address of the sender of a Commit Message. The length of the Anti-Clogging Token needs not be specified because the generation and processing of the Anti-Clogging Token is solely up to one peer. To the other peer in the SAE protocol, the Anti-Clogging Token is merely an opaque blob whose length is insignificant. It is suggested that an Anti-Clogging Token not exceed 256 octets.

NOTE—A suggested method for producing Anti-Clogging Tokens is to generate a random secret value each time the state machine variable hits dot11RSNASAEAntiCloggingThreshold and pass that secret and the MAC address of the sender of the Commit Message to the random function H to generate the token.

As long as the state machine variable *Open* is greater than dot11RSNASAEAntiCloggingThreshold all Commit Messages that do not include an Anti-Clogging Token must be rejected with a request to repeat the Commit Message and include the token (see 8.2a.6.1).

Since the Anti-Clogging Token is a fixed size and the size of the *peer-commit-scalar* and ***PEER-COMMIT-ELEMENT***(Ed) can be inferred from the finite cyclic group being used, it is straightforward to determine whether a received Commit Message includes an Anti-Clogging Token or not.

Encoding of the Anti-Clogging Token and its placement with respect to the *peer-commit-scalar* and ***PEER-COMMIT-ELEMENT***(Ed) is described in 8.2a.7.4 (Encoding and decoding of Commit Messages).

***Modify section 10.3.73.2.2 as indicated***

* Semantics of the service primitive

The primitive parameters are as follows:

MLME-MeshPeeringManagement.confirm(

 peerMAC,

 ResultCode,

 MeshPeeringMgmtFrameContent

 )

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Type | Valid range | Description |
| peerMAC | MAC Address | Valid individual MAC address | Specifies the address of the peer MAC entity to which the Mesh Peering Management frame was sent. |
| ResultCode | Enumeration | SUCCESS, TIMEOUT, INVALID\_PARAMETERS, or UNSPECIFIED\_FAILURE | Reports the outcome of the request to send a Mesh Peering Management frame. |
| MeshPeeringMgmtFrameContent | Sequence of octets | As defined in 7.4.14.2 (Mesh Peering Open frame format), 7.4.14.3 (Mesh Peering Confirm frame format), or 7.4.14.4 (Mesh Peering Close frame format). | The contents of the Action field of the Mesh Peering Open, Mesh Peering Confirm, or Mesh Peering Close frame received from the peer MAC entity. |

 **References:**